# 尚马教育 JAVA 基础课程

# 多线程

文档编号：A16

创建日期： 2017-04-12

最后修改日期：2019-09-23

版 本 号：V3.0

电子版文件名：尚马教育-第一阶段-16.多线程专题课程.docx

**文档修改记录：**

|  |  |  |  |
| --- | --- | --- | --- |
| 更新日期 | 更新作者 | 更新说明 | 版本号 |
| 2017-07-30 | 张元林 | 初始版本 | V1.0 |
| 2018-08-01 | 王绍成 | Java基础版本更新 | V2.0 |
| 2019-08-09 | 徐丽莎 | Java基础版本更新 | V3.0 |

**主讲人：**

**徐丽莎**

目录

[尚马教育 JAVA 基础课程 1](#_Toc26716)

[多线程 1](#_Toc24373)

[1. 术语解释 3](#_Toc4553)

[1.1. 并行 vs 并发 3](#_Toc14815)

[1.1.1. 并行 3](#_Toc12952)

[1.1.2. 并发 3](#_Toc9258)

[1.2. 同步 vs 异步 3](#_Toc28721)

[1.2.1. 同步 3](#_Toc19994)

[1.2.2. 异步 4](#_Toc16851)

[1.3. 阻塞 vs 非阻塞 4](#_Toc1016)

[1.3.1. 阻塞 4](#_Toc21644)

[1.3.2. 非阻塞 4](#_Toc4342)

[1.4. 组合使用 4](#_Toc14260)

[2. 线程(Thread) 5](#_Toc15418)

[2.1. 进程 vs 线程 5](#_Toc24158)

[2.1.1. 进程 5](#_Toc23292)

[2.1.2. 线程 5](#_Toc9584)

[2.1.3. 两者关系 5](#_Toc2267)

[2.2. Thread 6](#_Toc19251)

[2.2.1. 常用构造方法 6](#_Toc24471)

[2.2.2. 常用功能方法 6](#_Toc27796)

[2.3. 创建多线程方式 7](#_Toc4558)

[2.3.1. 继承Thread 7](#_Toc32130)

[2.3.2. 实现Runnable 9](#_Toc10350)

[2.3.3. 实现Callable 10](#_Toc7423)

[2.3.4. 其它写法 12](#_Toc24831)

[2.4. 线程安全 13](#_Toc27351)

[2.4.1. 自动加锁-->同步代码块 14](#_Toc23281)

[2.4.2. 自动加锁-->同步方法 15](#_Toc29741)

[2.4.3. 手动加锁-->Lock 16](#_Toc29834)

[2.4.4. synchornized vs Lock 17](#_Toc13617)

[2.4.5. 类锁 17](#_Toc31794)

[2.5. 死锁(了解) 18](#_Toc25020)

[2.6. 线程通信(协作) 20](#_Toc16684)

[2.6.1. 管程法 21](#_Toc29609)

[2.7. 线程状态 23](#_Toc9975)

[2.8. Thread类方法 24](#_Toc13608)

[2.8.1. 线程停止 24](#_Toc14582)

[3. 作业 25](#_Toc28245)

## 术语解释

### 并行 vs 并发

* 在同一时间内都可以执行多任务。

#### 并行

* 并行(Parallelism):是计算机系统中能同时执行两个或多个处理的一种计算方法.并行处理的主要目的是节省大型和复杂问题的解决时间.
* 比如：你吃饭吃到一半，电话来了，你一边打电话一边吃饭，这说明你支持并行。

#### 并发

* 并发(Concurrent):在操作系统中，是指一个时间段中有几个程序都处于已启动运行到运行完毕之间，两个或者多个任务都请求运行，而处理器只能按受一个任务，就把这两个或者多个任务安排轮流进行， 由于时间间隔较短，使人感觉两个任务都在运行。
* 一般场景大型网站，比如门户网站。在面对大量用户访问、高并发请求方面，基本的解决方案集中在这样几个环节：使用高性能的服务器、高性能的数据库、高效率的编程语言、还有高性能Web容器。但是除了这几个方面，还没法根本解决大型网站面临的高负载和高并发问题。
* 比如：你吃饭吃到一半，电话来了，你停了下来接了电话，接完后继续吃饭，这说明你支持并发。

### 同步 vs 异步

#### 同步

* 多个事物不能同时进行，必须一个一个的来，上一个事物结束后，下一个事物才开始.(在同一个时间内有且只有一个程序在执行)

#### 异步

* 多个事物同时进行。不用等待其它程序是否有返回的结果。(在后面学习ajax的时候更能体现异步-->一般称为异步请求)

### 阻塞 vs 非阻塞

#### 阻塞

* 指的是阻碍堵塞。它的本意可以理解为由于遇到了障碍而造成的动弹不得。

#### 非阻塞

* 自然是和阻塞相对，可以理解为由于没有遇到障碍而继续畅通无阻。
* 对这两个词最好的诠释就是，当今中国一大交通难题，堵车：汽车可以正常通行时，就是非阻塞。一旦堵上了，全部趴窝，一动不动，就是阻塞。因此阻塞关注的是不能动，非阻塞关注的是可以动。回到程序里，阻塞同样意味着停下来等待 wait，非阻塞表明可以继续向下执行

### 组合使用

* 所谓同步/异步，关注的是能不能同时开工。
* 所谓阻塞/非阻塞，关注的是能不能动。
  + 同步阻塞，不能同时开工，也不能动。只有一条小道，一次只能过一辆车，可悲的是还都堵上了。
  + 同步非阻塞，不能同时开工，但可以动。只有一条小道，一次只能过一辆车，幸运的是可以正常通行。
  + 异步阻塞，可以同时开工，但不可以动。有多条路，每条路都可以跑车，可气的是全都都的堵上了。
  + 异步非阻塞，可以同时开工，也可以动。有多条路，每条路都可以跑车，很爽的是全都可以正常通行。
* 回到程序里，把它们和线程关联起来：
  + 同步阻塞，相当于一个线程在等待。
  + 同步非阻塞，相当于一个线程在正常运行。
  + 异步阻塞，相当于多个线程都在等待。
  + 异步非阻塞，相当于多个线程都在正常运行。

想了解更多，可以访问这个地址：https://mp.weixin.qq.com/s/8ZR--B3Dg7VKrBgoIDRgeQ

## 线程(Thread)

### 进程 vs 线程

#### 进程

* 进程是资源（CPU、内存等）分配的基本单位，它是程序执行时的一个实例。程序运行时系统就会创建一个 进程，并为它分配资源，然后把该进程放入进程就绪队列，进程调度器选中它的时候就会为它分配CPU时间，程序开始真正运行。
* 简单来说，一个运行的程序就是一个进程。

#### 线程

* 线程是程序执行时的最小单位，它是进程的一个执行流，是CPU调度和分派的基本单位。一个进程可以由很多个线程组成，线程间共享进程的所有资源，每个线程有自己的堆栈和局部变量。线程由CPU独立调度执行，在多CPU环境下就允许多个线程同时运行。同样多线程也可以实现并发操作每个请求分配一个线程来处理。

#### 两者关系

* 进程是资源分配的最小单位，线程是程序执行的最小单位。
* 进程有自己的独立地址空间，每启动一个进程，系统就会为它分配地址空间，建立数据表来维护代码段、堆栈段和数据段，这种操作非常昂贵。线程是共享进程中的数据的，使用相同的地址空间，因此CPU切换一个线程的花费远比进程要小很多，同时创建一个线程的开销也比进程要小很多。
* 线程之间的通信更方便，同一进程下的线程共享全局变量、静态变量等数据。进程之间的通信需要以通信的方式（IPC)进行。不过如何处理好同步与互斥是编写多线程程序的难点。
* 但是多进程程序更健壮，多线程程序只要有一个线程死掉，整个进程也死掉了， 而一个进程死掉并不会对另外一个进程造成影响，因为进程有自己独立的地址空间。
* 一个进程有多个线程。

### Thread

#### 常用构造方法

* Thread类共提供8种构造方法重载，以下是常用的几种：

|  |
| --- |
|  |

#### 常用功能方法

|  |
| --- |
|  |
|  |

### 创建多线程方式

* Java语言，或者说jvm是支持多线程的。
* 优势：可以最大限度地减低CPU的闲置时间，从而提高CPU的利用率。

#### 继承Thread

* 案例：模拟多个线程同时下载网站资源

|  |
| --- |
| **package** com.javasm.thread; **import** com.javasm.demo.FileUtil; **public class** DownloadPictureThread **extends** Thread {  **private** String **pictureUrl**;//资源路径  **private** String **targetFileNme**;//目标文件名称  @Override  **public void** run() {  FileUtil.*downlodFile*(**pictureUrl**,**targetFileNme**);  System.***out***.println(**"成功下载了:"**+**targetFileNme**);  }  **public** DownloadPictureThread(String pictureUrl, String targetFileNme) {  this.**pictureUrl** = pictureUrl;  **this**.**targetFileNme** = targetFileNme;  } } |
| **package** com.javasm.util; **import** java.io.\*; **import** java.net.URL;  **public class** FileUtil {  **private** FileUtil() {  }  **public static void** downloadFile(String sourceFile, String targetFile) {  **try** (  InputStream bufferedInputStream = **new** BufferedInputStream(**new** URL(sourceFile).openStream());  OutputStream outputStream = **new** BufferedOutputStream(**new** FileOutputStream(**new** File(targetFile)));  ) {  **byte**[] bytes = **new byte**[1024 \* 10];  **int** len = 0;  **while** ((len = bufferedInputStream.read(bytes)) != -1) {  outputStream.write(bytes, 0, len);  }  } **catch** (FileNotFoundException e) {  e.printStackTrace();  } **catch** (IOException e) {  e.printStackTrace();  System.***out***.println(**"IO异常，文件下载失败"**);  }  } } |
| **package** com.javasm.test; **import** com.javasm.thread.DownloadPictureThread; **public class** ThreadTest {  **public static void** main(String[] args) {  DownloadPictureThread threadDemo1 = **new** DownloadPictureThread(**"https://www.javasm.cn/static/upload/image/20191231/1577783510242015.png"**,**"a.png"**);  DownloadPictureThread threadDemo2 = **new** DownloadPictureThread(**"https://www.javasm.cn/static/upload/image/20200219/1582127034814192.jpg"**,**"b.png"**);  DownloadPictureThread threadDemo3 = **new** DownloadPictureThread(**"https://www.javasm.cn/static/upload/image/20180702/1530526129232132.png"**,**"c.png"**);  threadDemo1.start();  threadDemo2.start();  threadDemo3.start();  }} |

#### 实现Runnable

* 案例：模拟烧水

|  |
| --- |
| **public** **class** BoilThreadRunnable **implements** Runnable { // 烧开水的线程  **public** **void** run() {  **try** {  System.***out***.println("开始烧水...");  Thread.*sleep*(10000); // 假设烧水需要10秒  System.***out***.println("水烧开了。");  } **catch** (InterruptedException ie) {  ie.printStackTrace();  }}} |
| **public** **class** WashThreadRunnable **implements** Runnable { // 洗茶杯的线程  **public** **void** run() {  **try** {  **for** (**int** i = 1; i <= 5; i++) { // 洗5个茶杯  System.***out***.print("开始洗第" + i + "个茶杯...");  Thread.*sleep*(1500); // 假设每洗一个茶杯需要1.5秒  System.***out***.println("第" + i + "个茶杯洗干净。");  }  } **catch** (InterruptedException ie) {  ie.printStackTrace();  }}} |
| **public** **class** MakeTea {  **public** **static** **void** main(String[] args) {  **new** Thread(**new** BoilThreadRunnable()).start(); // 启动烧水线程  **new** Thread(**new** WashThreadRunnable()).start(); // 启动洗茶杯线程  }  } |

#### 实现Callable

案例：模拟火车站窗口卖票

|  |
| --- |
| **public** **class** SaleTicketCallable **implements** Callable<String> {  // 重写run方法  **private** **int** ticketNum = 20;  @Override  **public** String call() {  **for** (**int** i = 1; i <= 20; i++) {  **if**(ticketNum>0) {  System.***out***.println(Thread.*currentThread*().getName() + "卖了第" + ticketNum + "张票");  ticketNum--;  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  **return** "";  }  } |
| **public** **class** SaleTicketTest2 {  **public** **static** **void** main(String[] args) {  // 创建3个线程 3个 窗口  SaleTicketCallable ticketCallable = **new** SaleTicketCallable();  // FutureTask  FutureTask<String> futureTask1 = **new** FutureTask<>(ticketCallable);  FutureTask<String> futureTask2 = **new** FutureTask<>(ticketCallable);  FutureTask<String> futureTask3 = **new** FutureTask<>(ticketCallable);  //结合线程池进行线程启动  //执行器 执行线程池里面的线程  ExecutorService executorService = Executors.newFixedThreadPool(3);  executorService.submit(futureTask1);  executorService.submit(futureTask2);  executorService.submit(futureTask3);  executorService.shutdown();//关闭池子  // Thread thread1 = new Thread(futureTask1);  // Thread thread2 = new Thread(futureTask2);  // Thread thread3 = new Thread(futureTask3);  // thread1.start();  // thread2.start();  // thread3.start();  //获得返回的数据  **try** {  System.***out***.println("result:"+futureTask1.get());  } **catch** (InterruptedException e) {  e.printStackTrace();  } **catch** (ExecutionException e) {  e.printStackTrace();  }  }  } |

#### 其它写法

|  |
| --- |
| **public** **static** **void** main(String[] args) {  **new** Thread() {  **public** **void** run() {  System.***out***.println("hello world");  }  }.start();  } |
| **public** **static** **void** main(String[] args) {  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  System.***out***.println("hello world");  }  }).start();  } |
| **public** **static** **void** main(String[] args) {  **new** Thread(()->{  System.***out***.println("hello world ");  }) .start();  } |

### 线程安全

* 在上述卖票的案例中，出现了卖票不对的情况 ：在多线程的环境下,多个线程访问同一个对象，或者说操作共同的资源，并且某些线程还要修改这个对象的数据。这时候我们就需要同步，线程同步其实就是一种等待机制。多个需要同时访问此对象的的线程进入这个对象的等待池形成队列，等待前面线程使用完毕，下一个线程再使用。
* 线程与线程之间相互独立。

|  |
| --- |
| **public** **class** SaleTicketRunnable **implements** Runnable {  // 重写run方法  **private** **int** ticketNum = 20;  @Override  **public** **void** run() {  **for** (**int** i = 1; i <= 20; i++) {  **if**(ticketNum>0) {  System.***out***.println(Thread.*currentThread*().getName() + "卖了第" + ticketNum + "张票");  ticketNum--;  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }}} |
| **public** **static** **void** main(String[] args) {  // 创建3个线程 3个 窗口  SaleTicketRunnable ticketRunnable1 = **new** SaleTicketRunnable();  Thread thread1 = **new** Thread(ticketRunnable1, "窗口1");  Thread thread2 = **new** Thread(ticketRunnable1, "窗口2");  Thread thread3 = **new** Thread(ticketRunnable1, "窗口3");  // 启动线程  thread1.start();  thread2.start();  thread3.start();  } |

#### 自动加锁-->同步代码块

|  |
| --- |
| **package** com.javasm.thread; **public class** SaleTicketRunnable **implements** Runnable {  **private int tickets** = 100;  **private boolean flag** = **true**;  @Override  **public void** run() {  **while**(**flag**){  **synchronized** (**"abc"**) {  **if** (**tickets**> 0) {  System.***out***.println(Thread.*currentThread*().getName() + **"在卖第"** + **tickets** + **"张票"**);  **tickets**--;  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }**else**{  flag = **false**;  }  }  }  } } |

#### 自动加锁-->同步方法

|  |
| --- |
| **package** com.javasm.thread; **public class** SaleTicketRunnable **implements** Runnable {  **private int tickets** = 100;  **private boolean flag** = **true**;  @Override  **public void** run() {  **while**(**flag**){  saleTiket();  }  }  **private synchronized void** saleTiket(){  **if** (**tickets**> 0) {  System.***out***.println(Thread.*currentThread*().getName() + **"在卖第"** + **tickets** + **"张票"**);  **tickets**--;  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }**else**{  **flag** = **false**;  }  } } |

#### 手动加锁-->Lock

* Lock： jdk5.0推出，提供了更加强大的同步机制。也就是显示定义同步锁对象来实现同步，同步锁对象使用Lock进行充当。
* Java.util.concurrent.locks.Lock接口是控制多个线程对共享资源进行访问的工具，锁提供了对共享资源的独占访问，每次只能有一个线程对Lock对象加锁，线程开始访问共享资源之前应该先获得Lock对象。
* ReentrantLock(可重入锁)类实现了Lock, 它拥有了与synchronized相同的并发性和内存语义，在实现线程安全的控制中，比较常用的就是这个实现类了。可以显示加锁，释放锁。

|  |
| --- |
| **public** **class** SaleTicketCallable **implements** Callable<String> {  **private** **int** ticketNum = 200;  **private** final Lock lock = **new** ReentrantLock();  @Override  **public** String call() **throws** Exception {  **for** (**int** i = 1; i <= 200; i++) {  lock.lock();//手动上锁  **if** (ticketNum > 0) {  System.***out***.println(Thread.*currentThread*().getName() + "卖了第" + ticketNum + "张票");  ticketNum--;  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  //手动释放锁 注意：有异常也需要释放 尽可能加上finally块  lock.unlock();  }  **return** "hello";  }  } |

#### synchornized vs Lock

* Lock是显示锁(手动开启和关闭)，synchronized是隐式锁，出了作用域就自动释放锁。
* Lock只能在方法体内使用，synchronized既可以在方法体内(对象锁)，也可以在方法上(类锁)。
* 使用Lock锁，JVM将花费更少的时间来调度线程，性能会更好，并且具有更好的可扩展性(有很多子类)。
* 优先使用顺序: Lock----> 同步代码块---->同步方法

#### 类锁

* 如果是继承Thread类的方式实现多线程，产生线程安全的时候，需要注意：

|  |
| --- |
| **public** **class** SaleTicketThread **extends** Thread {  **private** **static** **int** *ticketNum* = 20;  @Override  **public** **void** run() {  **for** (**int** i = 1; i <= 20; i++) {  *salTicket*();  }  }  **private** **synchronized** **static** **void** salTicket() {// 类锁 SaleTicketThread.class  **if** (*ticketNum* > 0) {  System.***out***.println(Thread.*currentThread*().getName() + "卖了第" + *ticketNum* + "张票");  *ticketNum*--;  }  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  **public** SaleTicketThread(String name) {  **super**(name);  }  } |

### 死锁(了解)

* 多线程的环境下
* 至少有2个资源
* 至少有2把锁

|  |
| --- |
| 手动演示: 嵌套锁+交叉锁  小明: 手里面有成绩单，想要父亲的零花钱  父亲：手里面有零花钱， 想要小明的成绩单  小明/父亲各自都有一把锁 ，锁住自己的资源 |
| **public** **class** Locks {  **public** **static** **final** String ***XIAOMING\_LOCK*** = "小明";  **public** **static** **final** String ***FATHER\_LOCK*** = "父亲";  } |
| **public** **class** XiaoMingThread **extends** Thread{  @Override  **public** **void** run() {  **synchronized** (Locks.XIAOMING\_LOCK) {//上锁  System.***out***.println("小明有成绩单");  **synchronized** (Locks.FATHER\_LOCK) {  System.***out***.println("小明想要父亲的零花钱");  }  }//解锁  }  } |
| **public** **class** FatherThread **extends** Thread{  @Override  **public** **void** run() {  **synchronized** (Locks.FATHER\_LOCK) {//上锁  System.***out***.println("父亲有零花钱");  **synchronized** (Locks.XIAOMING\_LOCK) {  System.***out***.println("父亲想要小明的成绩单");  }  }//解锁  }  } |
| **public** **class** DeadLockTest {  **public** **static** **void** main(String[] args) {  XiaoMingThread mingThread = **new** XiaoMingThread();  FatherThread fatherThread = **new** FatherThread();  mingThread.start();  fatherThread.start();  }  } |

### 线程通信(协作)

生产者与消费者模式，只是一种现象，一种问题。

* 什么是生产者?
  + 生产者指的是负责生产数据的模块(这里模块可能是：方法、对象、线程、进程)。
* 什么是消费者?
  + 消费者指的是负责处理数据的模块(这里模块可能是：方法、对象、线程、进程)。
* 什么是缓冲区?
  + 消费者不能直接使用生产者的数据，它们之间有个“缓冲区”。生产者将生产好的数据放入“缓冲区”，消费者从“缓冲区”拿要处理的数据

以上这种写法，我们也称为并发协作模型之管程法。

|  |
| --- |
|  |

#### 管程法

|  |
| --- |
| **import** java.util.concurrent.CopyOnWriteArrayList; **public class** OrderPool {  **private** CopyOnWriteArrayList<String> **list** = **new** CopyOnWriteArrayList<>();  //生产者放入订单  **public synchronized void** produceOrder(){  **if**(30 == **list**.size()){  **try** {  **this**.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  //如果没有满就将订单放入订单池  **list**.add(**"order"**);  System.***out***.println(Thread.*currentThread*().getName()+**"-------------->生产1个，还有"**+**list**.size()+**"个"**);  **try** {  Thread.*sleep*(300);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  //有订单就通知消费者消费了  **this**.notifyAll();  }  //消费者消费订单  **public synchronized void** consumerOrder(){  **if** (**list**.size()==0){ // System.out.println("订单池没有订单，消费者等待");  **try** {  **this**.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  //订单池子里面有订单 就消费 消费第一个  **list**.remove(0);  System.***out***.println(Thread.*currentThread*().getName()+**"-------------->消费1个，还有"**+**list**.size()+**"个"**);  **try** {  Thread.*sleep*(400);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  //消费完了 池子里面就有多余的空间 让生产者生产  **this**.notifyAll();  } } |
| **package** com.javasm.thread; **public class** ProduceOrder **extends** Thread {  **private** OrderPool **orderPool**;  **public** ProduceOrder(OrderPool orderPool,String name){  **super**(name);  **this**.**orderPool** = orderPool;  }  @Override  **public void** run() {  **while**(**true**){  **orderPool**.produceOrder();  }  } } |
| **package** com.javasm.thread; **public class** ConsumerOrder extends Thread {  **private** OrderPool **orderPool**;  **public** ConsumerOrder(OrderPool orderPool,String name){  **super**(name);  **this**.**orderPool** = orderPool;  }  @Override  **public void** run() {  **while** (**true**){  **orderPool**.consumerOrder();  }  } } |

### 线程状态

* 新建状态（New）：使用new关键字创建线程对象，仅仅被分配了内存；
* 就绪状态（Ready）：线程对象被创建后，等待它的start方法被调用，以获得CPU的使用权；
* 运行状态（Running）：执行run方法，此时的线程的对象正占用CPU；
* 睡眠状态（Sleeping）：调用sleep方法，线程被暂停，睡眠时间结束后，线程回到就绪状态，睡眠状态的线程不占用CPU；
* 死亡状态（Dead）：run方法执行完毕后，线程进入死亡状态；
* 阻塞状态（Blocked）：线程由于某些事件（如等待键盘输入）放弃CPU，暂停运行，直到线程重新进入就绪状态，才有机会转到运行状态；

|  |
| --- |
|  |
|  |

### 线程停止

在Thread类里面提供了stop()或者destory()等方法可以结束线程，但是不推荐使用，而且有的方法也已经过时了，在开发中，往往也不推荐使用过时的方法。

建议线程正常停止，利用次数。也可以使用标志位(变量)。

|  |
| --- |
| **package** com.javasm.thread; **public class** StopThread **implements** Runnable {  //1.通过变量(标志位)进行维护  **private boolean flag**;//默认值false  @Override  **public void** run() {  **int** i = 0;  **while**(!**flag**){  System.***out***.println(**"StopThread.......run()"**+i++);  }  }  **public void** stop(){  **this**.**flag** = **true**;  }  **public static void** main(String[] args) {  StopThread stopThread = **new** StopThread();  **new** Thread(stopThread).start();  **for** (**int** i = 0; i < 1000; i++) {  System.***out***.println(**"main================="**+i);  **if**(i==500){  //2.改变变量的数据  stopThread.stop();  System.***out***.println(**"线程要停止了"**);  }  }  } |

## 作业

* 创建两个线程，每个线程打印出线程名字后再睡眠，给其它线程以执行的机会，主线程也要打印出线程名字后再睡眠，每个线程前后共睡眠5次。要求分别采用从Thread中继承和实现Runnable接口两种方式来实现程序。（即写两个程序）
* 结合生产者与消费者模式编写如下程序：生产者随机时间（0.5-1.5秒）产生随机的字符串，消费者由线程担任，假设每处理一个字符串需要2-3秒时间